The Big Book of PowerShell Error Handling

PowerShell.org

Table of Contents

ReadMe	1.1
About this Book	1.2
Introduction	1.3
PowerShell Error Handling Basics	1.4
Controlling Error Reporting Behavior and Intercepting Errors	1.5
Analysis of Error Handling Test Results	1.6
Putting it All Together	1.7
Afterword	1.8

Despite the title, this is actually a very small, concise book designed to help you understand how PowerShell generates and deals with errors. It's also designed to help you build the best possible error handling for your own scripts and functions, in just a few short lessons.

The Big Book of PowerShell Error Handling

by Dave Wyatt

Cover design by Nathan Vonnahme

Despite the title, this is actually a very small, concise book designed to help you understand how PowerShell generates and deals with errors. It's also designed to help you build the best possible error handling for your own scripts and functions, in just a few short lessons.

This guide is released under the Creative Commons Attribution-NoDerivs 3.0 Unported License. The authors encourage you to redistribute this file as widely as possible, but ask that you do not modify the document.

Getting the code Sample code, along with a spreadsheet documenting known exception class names, can be found at https://github.com/devops-collective-inc/big-book-of-powershell-error-handling/tree/master/attachments.

Was this book helpful? The author(s) kindly ask(s) that you make a tax-deductible (in the US; check your laws if you live elsewhere) donation of any amount to The DevOps Collective to support their ongoing work.

Check for Updates! Our ebooks are often updated with new and corrected content. We make them available in three ways:

- Our main, authoritative GitHub organization, with a repo for each book. Visit https://github.com/devops-collective-inc/
- Our GitBook page, where you can browse books online, or download as PDF, EPUB, or MOBI. Using the online reader, you can link to specific chapters. Visit https://www.gitbook.com/@devopscollective
- On LeanPub, where you can download as PDF, EPUB, or MOBI (login required), and "purchase" the books to make a donation to DevOps Collective. Visit https://leanpub.com/u/devopscollective

GitBook and LeanPub have slightly different PDF formatting output, so you can choose the one you prefer. LeanPub can also notify you when we push updates. Our main GitHub repo is authoritative; repositories on other sites are usually just mirrors used for the publishing

process. GitBook will usually contain our latest version, including not-yet-finished bits; LeanPub always contains the most recent "public release" of any book.

Introduction

Error handling in Windows PowerShell can be a complex topic. The goal of this book - which is fortunately not as "big" as the name implies - is to help clarify some of that complexity, and help you do a better and more concise job of handling errors in your scripts.

What is error handling?

When we say that a script "handles" an error, this means it reacts to the error by doing something other than the default behavior. In many programming and scripting languages, the default behavior is simply to output an error message and immediately crash. In PowerShell, it will also output an error message, but will often continue executing code after the error occurred.

Handling errors requires the script's author to anticipate where errors might occur, and to write code to intercept and analyze those errors if and when they happen. This can be a complex and sometimes frustrating topic, particularly in PowerShell. The purpose of this book is to show you the error handling tools PowerShell puts at your disposal, and how best to use them.

How this book is organized

Following this introduction, the book is broken up into four sections. The first two sections are written to assume that you know nothing about PowerShell error handling, and to provide a solid background on the topic. However, there's nothing new in these sections that isn't already covered by the PowerShell help files. If you're already fairly familiar with the ErrorRecord object and the various parameters / variables / statements that are related to reporting and handling errors, you may want to skip straight to sections 3 and 4.

Section 3 is an objective look at how PowerShell's error handling features actually behave, based on the results of some test code I wrote to put it through its paces. The idea was to determine whether there are any functional differences between similar approaches to handling errors ($error versus ErrorVariable, whether to use $_ or not in a catch block, etc.), all of which generated some strong opinions during and after the 2013 Scripting Games.

These tests reveal a couple of tricky bugs, particularly involving the use of ErrorVariable.

Section 4 wraps things up by giving you a more task-oriented view of error handling, taking the findings from section 3 into consideration.

Powershell Error Handling Basics

Let's start by getting some of the basics out of the way.

ErrorRecords and Exceptions

In the .NET Framework, on which PowerShell is built, error reporting is largely done by throwing exceptions. Exceptions are .NET Objects which have a base type of System.Exception.aspx). These Exception objects contain enough information to communicate all the details of the error to a .NET Framework application (the type of error that occurred, a stack trace of method calls that led to the error, etc.) That alone isn't enough information to provide to a PowerShell script, though; PowerShell has its own stack trace of scripts and function calls which the underlying .NET Framework knows nothing about. It's also important to know which objects had failures, when a single statement or pipeline is capable of producing multiple errors.

For these reasons, PowerShell gives us the ErrorRecord object. ErrorRecords contain a .NET Exception, along with several other pieces of PowerShell-specific information. For example, figure 1.1 shows how you can access the TargetObject, CategoryInfo and InvocationInfo properties of an ErrorRecord object; any one of these might provide information that is useful to your script's error handling logic.

```
PS C:\Source\temp> Get-Item C:\Does\Not\Exist.txt
Get-Item : Cannot find path 'C:\Does\Not\Exist.txt' because it does not exist.
At line:1 char:1
+ Get-Item C:\Does\Not\Exist.txt
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : ObjectNotFound: (C:\Does\Not\Exist.txt:String) [Get-Item], ItemNotFoundException
    + FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.GetItemCommand

PS C:\Source\temp> $error[0].TargetObject
C:\Does\Not\Exist.txt
PS C:\Source\temp> $error[0].CategoryInfo


Category   : ObjectNotFound
Activity   : Get-Item
Reason     : ItemNotFoundException
TargetName : C:\Does\Not\Exist.txt
TargetType : String



PS C:\Source\temp> $error[0].InvocationInfo


MyCommand             : Get-Item
BoundParameters       : {}
UnboundArguments      : {}
ScriptLineNumber      : 1
OffsetInLine          : 1
HistoryId             : 2
ScriptName            :
Line                  : Get-Item C:\Does\Not\Exist.txt
PositionMessage       : At line:1 char:1
                        + Get-Item C:\Does\Not\Exist.txt
                        + ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
PSScriptRoot          :
PSCommandPath         :
InvocationName        : Get-Item
PipelineLength        : 0
PipelinePosition      : 0
ExpectingInput        : False
CommandOrigin         : Internal
DisplayScriptPosition :



PS C:\Source\temp>
```

Figure 1.1: Some of the ErrorRecord object's more useful properties.

Terminating versus Non-Terminating Errors

PowerShell is an extremely *expressive* language. This means that a single statement or pipeline of PowerShell code can perform the work of hundreds, or even thousands of raw CPU instructions. For example:

Get-Content .\computers.txt | Restart-Computer

This small, 46-character PowerShell pipeline opens a file on disk, automatically detects its text encoding, reads the text one line at a time, connects to each remote computer named in the file, authenticates to that computer, and if successful, restarts the computer. Several of these steps might encounter errors; in the case of the Restart-Computer command, it may succeed for some computers and fail for others.

For this reason, PowerShell introduces the concept of a Non-Terminating error. A Non-Terminating error is one that does not prevent the command from moving on and trying the next item on a list of inputs; for example, if one of the computers in the computers.txt file is offline, that doesn't stop PowerShell from moving on and rebooting the rest of the computers in the file.

By contrast, a Terminating error is one that causes the entire pipeline to fail. For example, this similar command fetches the email addresses associated with Active Directory user accounts:

```
Get-Content .\users.txt |
Get-ADUser -Properties mail |
Select-Object -Property SamAccountName,mail
```

In this pipeline, if the Get-ADUser command can't communicate with Active Directory at all, there's no reason to continue reading lines from the text file or attempting to process additional records, so it will produce a Terminating error. When this Terminating error is encountered, the entire pipeline is immediately aborted; Get-Content will stop reading lines, and close the file.

It's important to know the distinction between these types of errors, because your scripts will use different techniques to intercept them. As a general rule, most errors produced by Cmdlets are non-terminating (though there are a few exceptions, here and there.)

Controlling Error Reporting Behavior and Intercepting Errors

This section briefly demonstrates how to use each of PowerShell's statements, variables and parameters that are related to the reporting or handling of errors.

The $Error Variable

$Error is an automatic global variable in PowerShell which always contains an ArrayList of zero or more ErrorRecord objects. As new errors occur, they are added to the beginning of this list, so you can always get information about the most recent error by looking at $Error[0]. Both Terminating and Non-Terminating errors will be contained in this list.

Aside from accessing the objects in the list with array syntax, there are two other common tasks that are performed with the $Error variable: you can check how many errors are currently in the list by checking the $Error.Count property, and you can remove all errors from the list with the $Error.Clear() method. For example:

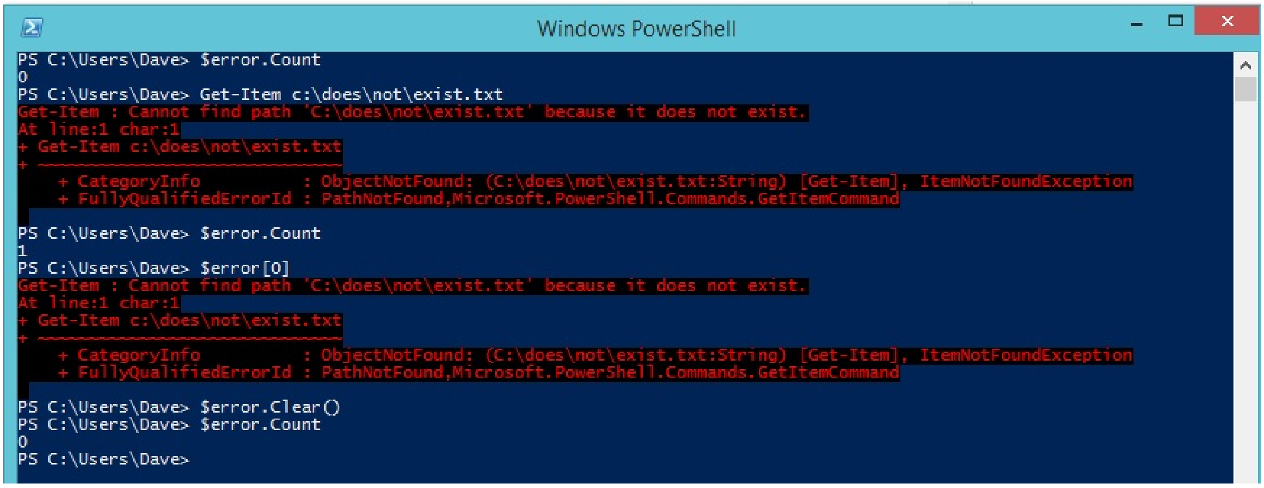


Figure 2.1: Using $Error to access error information, check the count, and clear the list.

If you're planning to make use of the $Error variable in your scripts, keep in mind that it may already contain information about errors that happened in the current PowerShell session before your script was even started. Also, some people consider it a bad practice to clear the $Error variable inside a script; since it's a variable global to the PowerShell session, the person that called your script might want to review the contents of $Error after it completes.

ErrorVariable

The ErrorVariable common parameter provides you with an alternative to using the built-in $Error collection. Unlike $Error, your ErrorVariable will only contain errors that occurred from the command you're calling, instead of potentially having errors from elsewhere in the current PowerShell session. This also avoids having to clear the $Error list (and the breach of etiquette that entails.)

When using ErrorVariable, if you want to append to the error variable instead of overwriting it, place a + sign in front of the variable's name. Note that you do not use a dollar sign when you pass a variable name to the ErrorVariable parameter, but you do use the dollar sign later when you check its value.

The variable assigned to the ErrorVariable parameter will never be null; if no errors occurred, it will contain an ArrayList object with a Count of 0, as seen in figure 2.2:

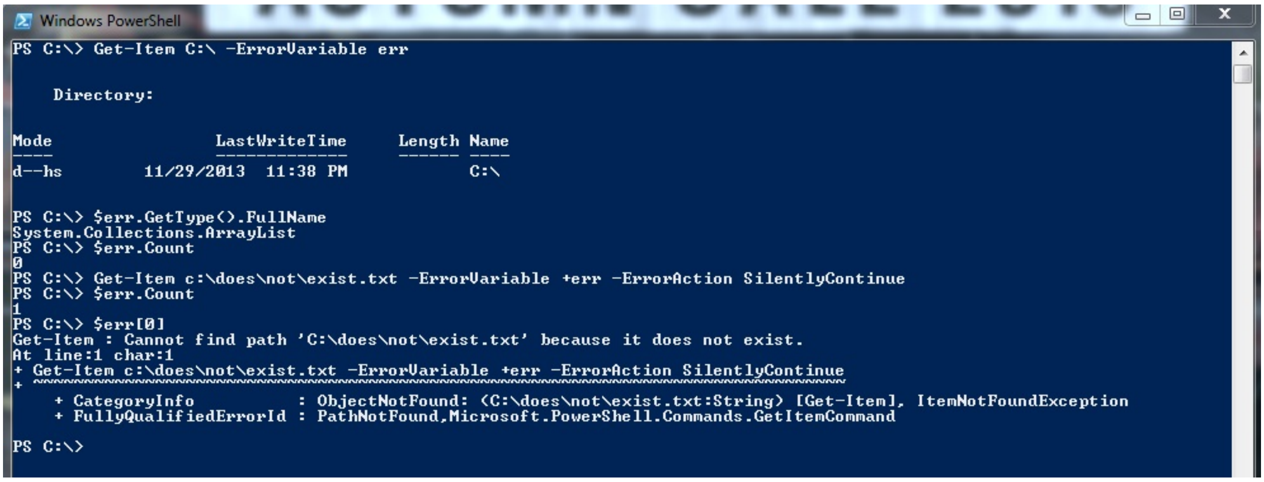


Figure 2.2: Demonstrating the use of the ErrorVariable parameter.

$MaximumErrorCount

By default, the $Error variable can only contain a maximum of 256 errors before it starts to lose the oldest ones on the list. You can adjust this behavior by modifying the $MaximumErrorCount variable.

ErrorAction and $ErrorActionPreference

There are several ways you can control PowerShell's handling / reporting behavior. The ones you will probably use most often are the ErrorAction common parameter and the $ErrorActionPreference variable.

The ErrorAction parameter can be passed to any Cmdlet or Advanced Function, and can have one of the following values: Continue (the default), SilentlyContinue, Stop, Inquire, Ignore (only in PowerShell 3.0 or later), and Suspend (only for workflows; will not be discussed further here.) It affects how the Cmdlet behaves when it produces a non-terminating error.

- The default value of Continue causes the error to be written to the Error stream and added to the $Error variable, and then the Cmdlet continues processing.
- A value of SilentlyContinue only adds the error to the $Error variable; it does not write the error to the Error stream (so it will not be displayed at the console).
- A value of Ignore both suppresses the error message and does not add it to the $Error variable. This option was added with PowerShell 3.0.
- A value of Stop causes non-terminating errors to be treated as terminating errors instead, immediately halting the Cmdlet's execution. This also enables you to intercept those errors in a Try/Catch or Trap statement, as described later in this section.
- A value of Inquire causes PowerShell to ask the user whether the script should continue or not when an error occurs.

The $ErrorActionPreference variable can be used just like the ErrorAction parameter, with a couple of exceptions: you cannot set $ErrorActionPreference to either Ignore or Suspend. Also, $ErrorActionPreference affects your current scope in addition to any child commands you call; this subtle difference has the effect of allowing you to control the behavior of errors that are produced by .NET methods, or other causes such as PowerShell encountering a "command not found" error.

Figure 2.3 demonstrates the effects of the three most commonly used $ErrorActionPreference settings.

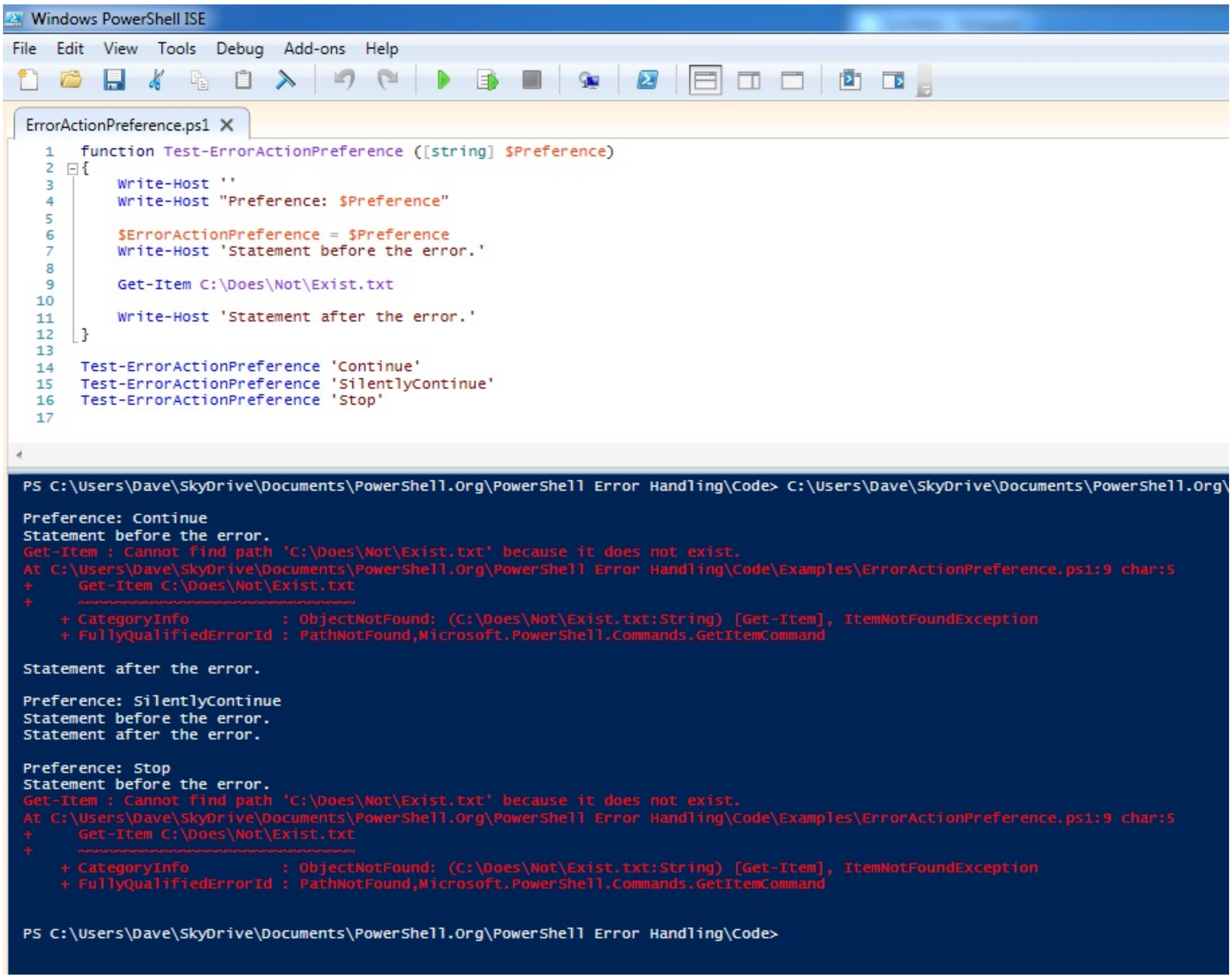


Figure 2.3: Behavior of $ErrorActionPreference

Try/Catch/Finally

The Try/Catch/Finally statements, added in PowerShell 2.0, are the preferred way of handling *terminating* errors. They cannot be used to handle non-terminating errors, unless you force those errors to become terminating errors with ErrorAction or $ErrorActionPreference set to Stop.

To use Try/Catch/Finally, you start with the "Try" keyword followed by a single PowerShell script block. Following the Try block can be any number of Catch blocks, and either zero or one Finally block. There must be a minimum of either one Catch block or one Finally block; a Try block cannot be used by itself.

The code inside the Try block is executed until it is either complete, or a terminating error occurs. If a terminating error does occur, execution of the code in the Try block stops. PowerShell writes the terminating error to the $Error list, and looks for a matching Catch

block (either in the current scope, or in any parent scopes.) If no Catch block exists to handle the error, PowerShell writes the error to the Error stream, the same thing it would have done if the error had occurred outside of a Try block.

Catch blocks can be written to only catch specific types of Exceptions, or to catch all terminating errors. If you do define multiple catch blocks for different exception types, be sure to place the more specific blocks at the top of the list; PowerShell searches catch blocks from top to bottom, and stops as soon as it finds one that is a match.

If a Finally block is included, its code is executed after both the Try and Catch blocks are complete, regardless of whether an error occurred or not. This is primarily intended to perform cleanup of resources (freeing up memory, calling objects' Close() or Dispose() methods, etc.)

Figure 2.4 demonstrates the use of a Try/Catch/Finally block:

TryCatchFinally.ps1

```powershell
try
{
    $testVariable = 'This is a test.'

    Write-Host 'Statement before the error.'

    [System.IO.File]::ReadAllText('C:\does\not\exist.txt')

    Write-Host 'Statement after the error.'
}
catch [System.IO.IOException]
{
    Write-Host 'An IOException was caught.'
    Write-Host "Exception type: $($_.Exception.GetType().FullName)"
}
catch
{
    Write-Host 'Some other type of error was caught.'
}
finally
{
    $testVariable = 'The finally block was executed.'
}

Write-Host "`$testVariable = '$testVariable'."

```

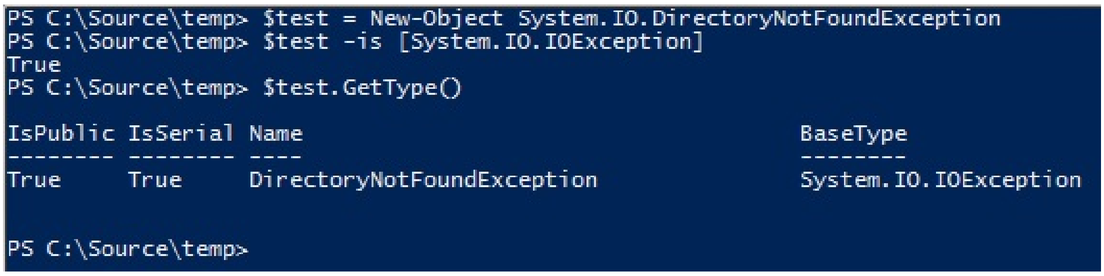
```
PS C:\Users\dwyatt> C:\Source\PowerShell\ErrorHandling\TryCatchFinally.ps1
Statement before the error.
An IOException was caught.
Exception type: System.IO.DirectoryNotFoundException
$testVariable = 'The finally block was executed.'.

PS C:\Users\dwyatt>
```

Figure 2.4: Example of using try/catch/finally.

Notice that "Statement after the error" is never displayed, because a terminating error occurred on the previous line. Because the error was based on an IOException, that Catch block was executed, instead of the general "catch-all" block below it. Afterward, the Finally executes and changes the value of $testVariable.

Also notice that while the Catch block specified a type of [System.IO.IOException], the actual exception type was, in this case, [System.IO.DirectoryNotFoundException]. This works because DirectoryNotFoundException is *inherited* from IOException, the same way all exceptions share the same base type of System.Exception. You can see this in figure 2.5:



```
PS C:\Source\temp> $test = New-Object System.IO.DirectoryNotFoundException
PS C:\Source\temp> $test -is [System.IO.IOException]
True
PS C:\Source\temp> $test.GetType()

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     DirectoryNotFoundException               System.IO.IOException


PS C:\Source\temp>
```

Figure 2.5: Showing that IOException is the Base type for DirectoryNotFoundException

Trap

Trap statements were the method of handling terminating errors in PowerShell 1.0. As with Try/Catch/Finally, the Trap statement has no effect on non-terminating errors.

Trap is a bit awkward to use, as it applies to the entire scope where it is defined (and child scopes as well), rather than having the error handling logic kept close to the code that might produce the error the way it is when you use Try/Catch/Finally. For those of you familiar with Visual Basic, Trap is a lot like "On Error Goto". For that reason, Trap statements don't see a lot of use in modern PowerShell scripts, and I didn't include them in the test scripts or analysis in Section 3 of this ebook.

For the sake of completeness, here's an example of how to use Trap:

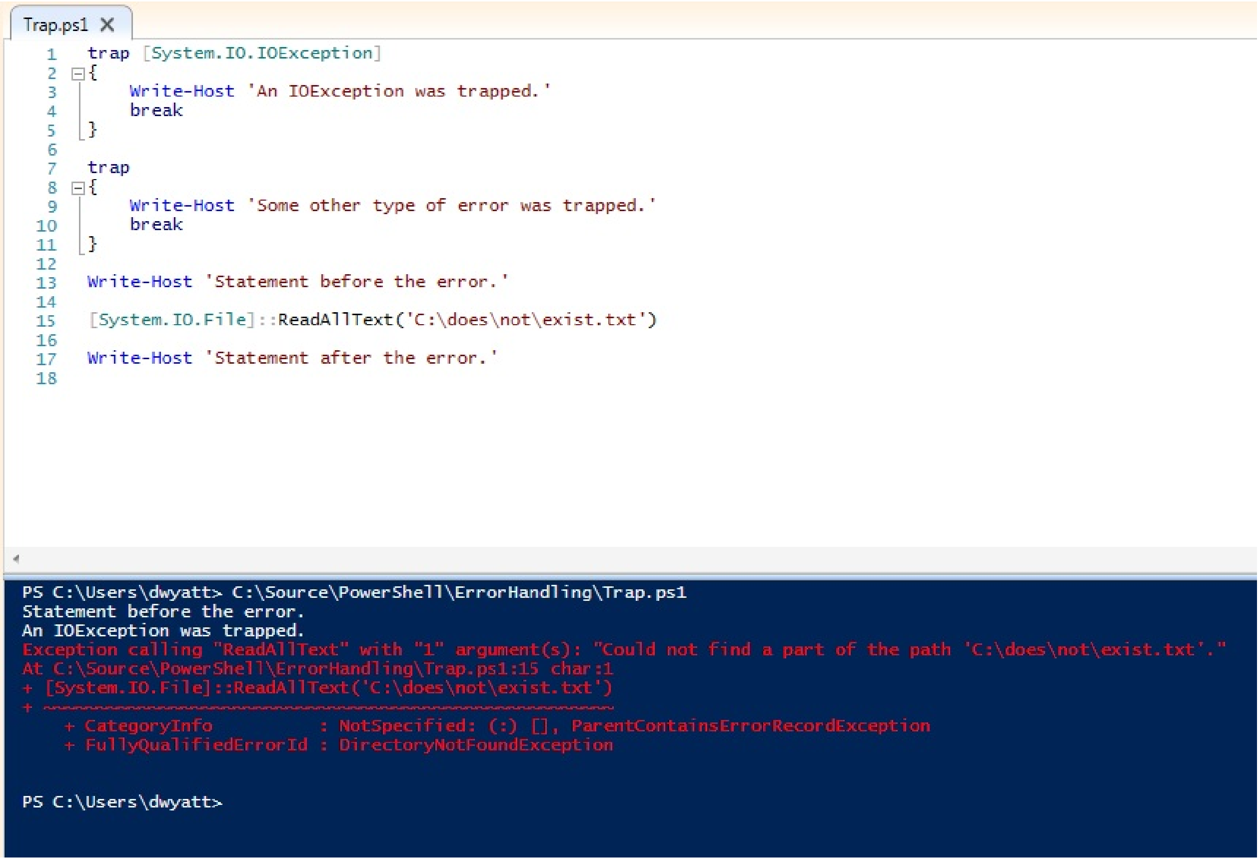


Figure 2.6: Use of the Trap statement

As you can see, Trap blocks are defined much the same way as Catch blocks, optionally specifying an Exception type. Trap blocks may optionally end with either a Break or Continue statement. If you don't use either of those, the error is written to the Error stream, and the current script block continues with the next line after the error. If you use Break, as seen in figure 2.5, the error is written to the Error stream, and the rest of the current script block is not executed. If you use Continue, the error is not written to the error stream, and the script block continues execution with the next statement.

The $LASTEXITCODE Variable

When you call an executable program instead of a PowerShell Cmdlet, Script or Function, the $LASTEXITCODE variable automatically contains the process's exit code. Most processes use the convention of setting an exit code of zero when the code finishes successfully, and non-zero if an error occurred, but this is not guaranteed. It's up to the developer of the executable to determine what its exit codes mean.

Note that the $LASTEXITCODE variable is only set when you call an executable directly, or via PowerShell's call operator (&) or the Invoke-Expression cmdlet. If you use another method such as Start-Process or WMI to launch the executable, they have their own ways of

communicating the exit code to you, and will not affect the current value of $LASTEXITCODE.

```
PS C:\> cmd.exe /c "dir c:\*.bat"
 Volume in drive C is Local Disk
 Volume Serial Number is 7859-839E

 Directory of c:\

06/10/2009  04:42 PM                24 autoexec.bat
               1 File(s)             24 bytes
               0 Dir(s)  91,893,669,888 bytes free
PS C:\> $LASTEXITCODE
0
PS C:\> ping.exe 10.0.0.2 -n 1

Pinging 10.0.0.2 with 32 bytes of data:
Request timed out.

Ping statistics for 10.0.0.2:
    Packets: Sent = 1, Received = 0, Lost = 1 (100% loss),
PS C:\> $LASTEXITCODE
1
PS C:\>
```

Figure 2.7: Using $LASTEXITCODE.

The $? Variable

The $? variable is a Boolean value that is automatically set after each PowerShell statement or pipeline finishes execution. It should be set to True if the previous command was successful, and False if there was an error. If the previous command was a call to a native exe, $? will be set to True if the $LASTEXITCODE variable equals zero, and False otherwise. When the previous command was a PowerShell statement, $? will be set to False if any errors occurred (even if ErrorAction was set to SilentlyContinue or Ignore.)

Just be aware that the value of this variable is reset after every statement. You must check its value immediately after the command you're interested in, or it will be overwritten (probably to True). Figure 2.8 demonstrates this behavior. The first time $? is checked, it is set to False, because the Get-Item encountered an error. The second time $? was checked, it was set to True, because the previous command was successful; in this case, the previous command was "$?" from the first time the variable's value was displayed.

```
PS C:\> Get-Item c:\does\not\exist.txt -ErrorAction Ignore
PS C:\> $?
False
PS C:\> $?
True
PS C:\> _
```

Figure 2.8: Demonstrating behavior of the $? variable.

The $? variable doesn't give you any details about what error occurred; it's simply a flag that something went wrong. In the case of calling executable programs, you need to be sure that they return an exit code of 0 to indicate success and non-zero to indicate an error before you

can rely on the contents of $?.

Summary

That covers all of the techniques you can use to either control error reporting or intercept and handle errors in a PowerShell script. To summarize:

- To intercept and react to non-terminating errors, you check the contents of either the automatic $Error collection, or the variable you specified as the ErrorVariable. This is done after the command completes; you cannot react to a non-terminating error before the Cmdlet or Function finishes its work.
- To intercept and react to terminating errors, you use either Try/Catch/Finally (preferred), or Trap (old and not used much now.) Both of these constructs allow you to specify different script blocks to react to different types of Exceptions.
- Using the ErrorAction parameter, you can change how PowerShell cmdlets and functions report non-terminating errors. Setting this to Stop causes them to become terminating errors instead, which can be intercepted with Try/Catch/Finally or Trap.
- $ErrorActionPreference works like ErrorAction, except it can also affect PowerShell's behavior when a terminating error occurs, even if those errors came from a .NET method instead of a cmdlet.
- $LASTEXITCODE contains the exit code of external executables. An exit code of zero usually indicates success, but that's up to the author of the program.
- $? can tell you whether the previous command was successful, though you have to be careful about using it with external commands, if they don't follow the convention of using an exit code of zero as an indicator of success. You also need to make sure you check the contents of $? immediately after the command you are interested in.

Analysis of Error Handling Test Results

As mentioned in the introduction, the test code and its output files are available for download. See the "About this Book" section, at the start of this book, for the location. It's quite a bit of data, and doesn't format very well in a Word document, so I won't be including the contents of those files in this ebook. If you question any of the analysis or conclusions I've presented in this section, I encourage you to download and review both the code and results files.

The test code consists of two files. The first is a PowerShell script module (ErrorHandlingTestCommands.psm1) which contains a Cmdlet, a .NET class and several Advanced Functions for producing terminating and non-terminating errors on demand, or for testing PowerShell's behavior when such errors are produced. The second file is the ErrorTests.ps1 script, which imports the module, calls its commands with various parameters, and produces output that was redirected (including the Error stream) to the three results files: ErrorTests.v2.txt, ErrorTests.v3.txt and ErrorTests.v4.txt.

There are three main sections to the ErrorTests.ps1 script. The first section calls commands to generate terminating and non-terminating errors, and outputs information about the contents of $_ (in Catch blocks only), $Error, and ErrorVariable. These tests were aimed at answering the following questions:

- When dealing only with non-terminating errors, are there differences between how $Error and ErrorVariable present information about the errors that occurred? Does it make any difference if the errors came from a Cmdlet or Advanced Function?
- When using a Try/Catch block, are there any differences in behavior between the way $Error, ErrorVariable, and $_ give information about the terminating error that occurred? Does it make any difference if the errors came from a Cmdlet, Advanced Function, or .NET method?
- When non-terminating errors happened in addition to the terminating error, are there differences between how $Error and ErrorVariable present the information? Does it make any difference if the errors came from a Cmdlet or Advanced Function?
- In the above tests, are there any differences between a terminating error that was produced normally, as compared to a non-terminating error that occurred when ErrorAction or $ErrorActionPreference was set to Stop?

The second section consists of a few tests to determine whether ErrorAction or $ErrorActionPreference affect terminating errors, or only non-terminating errors.

The final section tests how PowerShell behaves when it encounters unhandled terminating errors from each possible source (a Cmdlet that uses PSCmdlet.ThrowTerminatingError(), an Advanced Function that uses PowerShell's Throw statement, a .NET method that throws an exception, a Cmdlet or Advanced Function that produce non-terminating errors when ErrorAction is set to Stop, and an unknown command.)

The results of all tests were identical in PowerShell 3.0 and 4.0. PowerShell 2.0 had a couple of differences, which will be called out in the analysis.

Intercepting Non-Terminating Errors

Let's start by talking about non-terminating errors.

ErrorVariable versus $Error

When dealing with non-terminating errors, there is only one difference between $Error and ErrorVariable: the order of errors in the lists is reversed. The most recent error that occurred is always at the beginning of the $Error variable (index zero), and the most recent error is at the end of the ErrorVariable.

Intercepting Terminating Errors

This is the real meat of the task: working with terminating errors, or exceptions.

$_

At the beginning of a Catch block, the $_ variable always refers to an ErrorRecord object for the terminating error, regardless of how that error was produced.

$Error

At the beginning of a Catch block, $Error[0] always refers to an ErrorRecord object for the terminating error, regardless of how that error was produced.

ErrorVariable

Here, things start to get screwy. When a terminating error is produced by a cmdlet or function and you're using ErrorVariable, the variable will contain some unexpected items, and the results are quite different across the various tests performed:

- When calling an Advanced Function that throws a terminating error, the ErrorVariable contains two identical ErrorRecord objects for the terminating error.In addition, if you're running PowerShell 2.0, these ErrorRecords are followed by two identical objects of type System.Management.Automation.RuntimeException. These RuntimeException objects contain an ErrorRecord property, which refers to ErrorRecord objects identical to the pair that was also contained in the ErrorVariable list. The extra RuntimeException objects are not present in PowerShell 3.0 or later.
- When calling a Cmdlet that throws a terminating error, the ErrorVariable contains a single record, but is not an ErrorRecord object. Instead, it's an instance of System.Management.Automation.CmdletInvocationException. Like the RuntimeException objects mentioned in the last point, CmdletInvocationException contains an ErrorRecord property, and that property refers to the ErrorRecord object that you would have expected to be contained in the ErrorVariable list.
- When calling an Advanced Function with ErrorAction set to Stop, the ErrorVariable contains one object of type System.Management.Automation.ActionPreferenceStopException, followed by two identical ErrorRecord objects. As with the RuntimeException and CmdletInvocationException types, ActionPreferenceStopException contains an ErrorRecord property, which refers to an ErrorRecord object that is identical to the two that were included directly in the ErrorVariable's list.In addition, if running PowerShell 2.0, there are then two more identical objects of type ActionPreferenceStopException, for a total of 5 entries all related to the same terminating error.
- When calling a Cmdlet with ErrorAction set to Stop, the ErrorVariable contains a single object of type System.Management.Automation.ActionPreferenceStopException. The ErrorRecord property of this ActionPreferenceStopException object contains the ErrorRecord object that you would have expected to be directly in the ErrorVariable's list.

Effects of setting ErrorAction or $ErrorActionPreference

When you execute a Cmdlet or Advanced Function and set the ErrorAction parameter, it affects the behavior of all non-terminating errors. However, it also appears to affect terminating errors produced by the Throw statement in an Advanced Function (though not terminating errors coming from Cmdlets via the PSCmdlet.ThrowTerminatingError() method.)

If you set the $ErrorActionPreference variable before calling the command, its value affects both terminating and non-terminating errors.

This is undocumented behavior; PowerShell's help files state that both the preference variable and parameter should only be affecting non-terminating errors.

How PowerShell behaves when it encounters unhandled terminating errors

This section of the code proved to be a bit annoying to test, because if the parent scope (the script) handled the errors, it affected the behavior of the code inside the functions. If the script scope didn't have any error handling, then in many cases, the unhandled error actually aborted the script as well. As a result, the ErrorTests.ps1 script and the text files containing its output are written to only show you the cases where a terminating error occurs, but the function still moves on and executes the next command.

If you want to run the full battery of tests on this behavior, import the ErrorHandlingTests.psm1 module and execute the following commands manually at a PowerShell console. Because you're executing them one at a time, you won't run into an issue with some of the commands failing to execute because of a previous unhandled error, the way you would if these were all in a script.

```
Test-WithoutRethrow -Cmdlet -Terminating

Test-WithoutRethrow -Function -Terminating

Test-WithoutRethrow -Cmdlet -NonTerminating

Test-WithoutRethrow -Function -NonTerminating

Test-WithoutRethrow -Method

Test-WithoutRethrow -UnknownCommand
```

There is also a Test-WithRethrow function that can be called with the same parameters, to demonstrate that the results are consistent across all 6 cases when you handle each error and choose whether to abort the function.

PowerShell continues execution after a terminating error is produced by:

- Terminating errors from Cmdlets.
- .NET Methods that throw exceptions.
- PowerShell encountering an unknown command.

PowerShell aborts execution when a terminating error is produced by:

- Functions that use the Throw statement.
- Any non-terminating error in conjunction with ErrorAction Stop.
- Any time $ErrorActionPreference is set to Stop in the caller's scope.

In order to achieve consistent behavior between these different sources of terminating errors, you can put commands that might potentially produce a terminating error into a Try block. In the Catch block, you can decide whether to abort execution of the current script block or not. Figure 3.1 shows an example of forcing a function to abort when it hits a terminating exception from a Cmdlet (a situation where PowerShell would normally just continue and execute the "After terminating error." statement), by re-throwing the error from the Catch block. When Throw is used with no arguments inside of a Catch block, it passes the same error up to the parent scope.

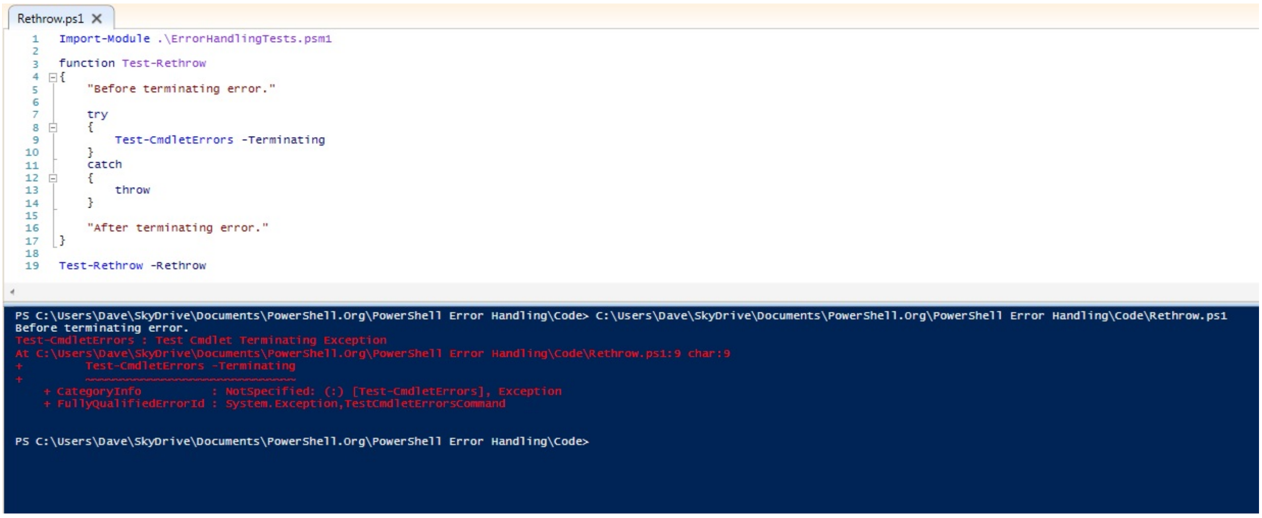


Figure 3.1: Re-throwing a terminating error to force a function to stop execution.

Conclusions

For non-terminating errors, you can use either $Error or ErrorVariable without any real headaches. While the order of the ErrorRecords is reversed between these options, you can easily deal with that in your code, assuming you consider that to be a problem at all. As soon as terminating errors enter the picture, however, ErrorVariable has some very annoying behavior: it sometimes contains Exception objects instead of ErrorRecords, and in many cases, has one or more duplicate objects all relating to the terminating error. While it is possible to code around these quirks, it really doesn't seem to be worth the effort when you can easily use $_ or $Error[0].

When you're calling a command that might produce a terminating error and you do not handle that error with a Try/Catch or Trap statement, PowerShell's behavior is inconsistent, depending on how the terminating error was generated. In order to achieve consistent results regardless of what commands you're calling, place such commands into a Try block, and choose whether or not to re-throw the error in the Catch block.

Putting It All Together

Now that we've looked at all of the error handling tools and identified some potential "gotcha" scenarios, here are some tips and examples of how I approach error handling in my own scripts.

Suppressing errors (Mostly, don't do this)

There are occasions where you might suppress an error without the intention of handling it, but the valid situations for this are few and far between. For the most part, don't set ErrorAction or $ErrorActionPreference to SilentlyContinue unless you intend to examine and respond to the errors yourself later in the code. Using Try/Catch with an empty catch block amounts to the same thing for terminating errors; it's usually the wrong thing to do.

It's better to at least give the user the default error output in the console than it is to have a command fail with no indication whatsoever that something went wrong.

The $? variable (Use it at your own risk)

The $? variable seems like a good idea on paper, but there are enough ways for it to give you bad data that I just don't trust it in a production script. For example, if the error is generated by a command that is in parentheses or a sub-expression, the $? variable will be set to True instead of False:

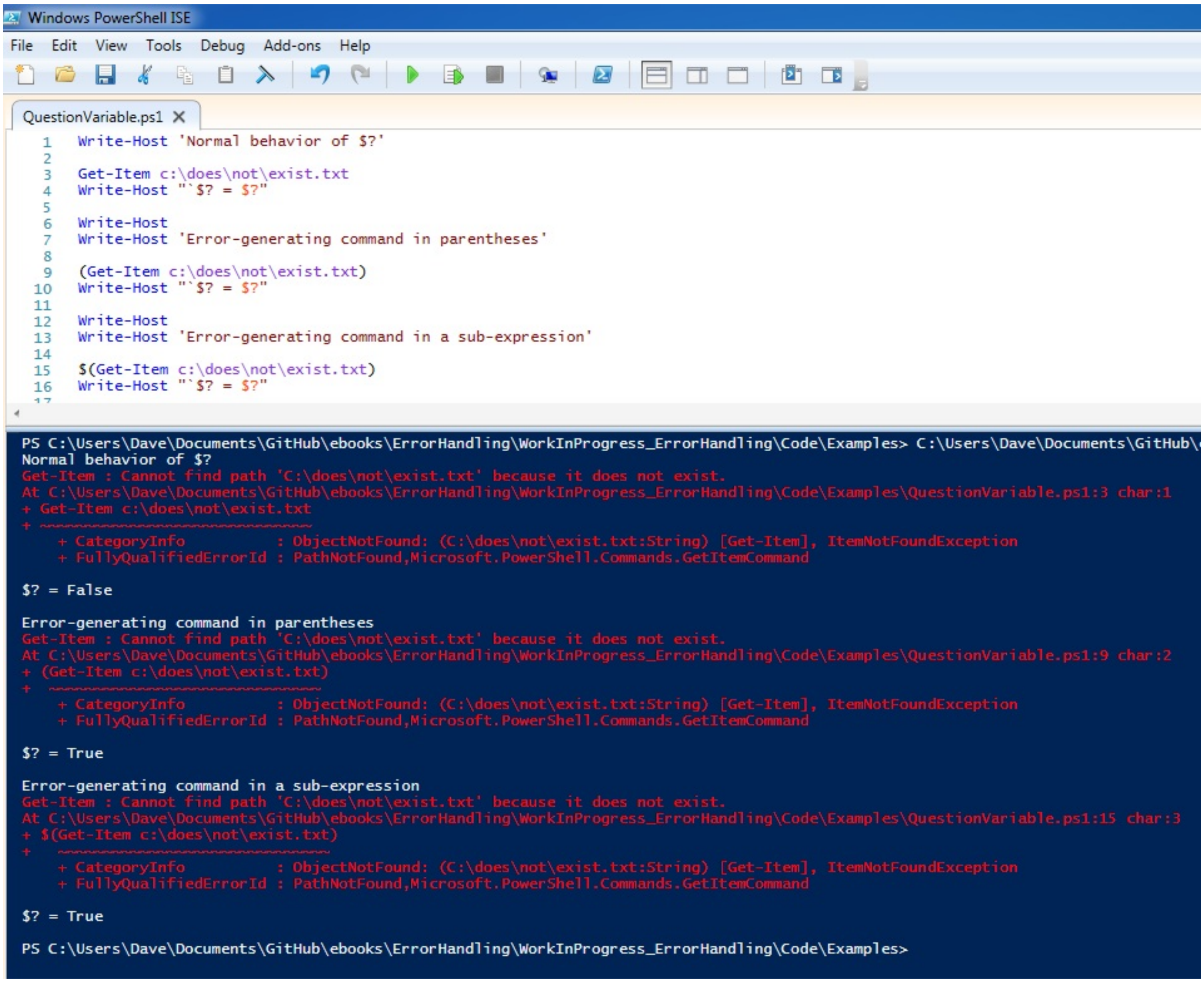


Figure 4.1: Annoying false positives from $?

Determining what types of errors can be produced by a command

Before you can decide how best to handle the error(s) from a particular command, you'll often need to know what kind of errors it might produce. Are they terminating or non-terminating? What are the Exception types? Unfortunately, PowerShell's cmdlet documentation doesn't give you this information, so you need to resort to some trial and error. Here's an example of how you can figure out whether errors from a cmdlet are Terminating or Non-Terminating:

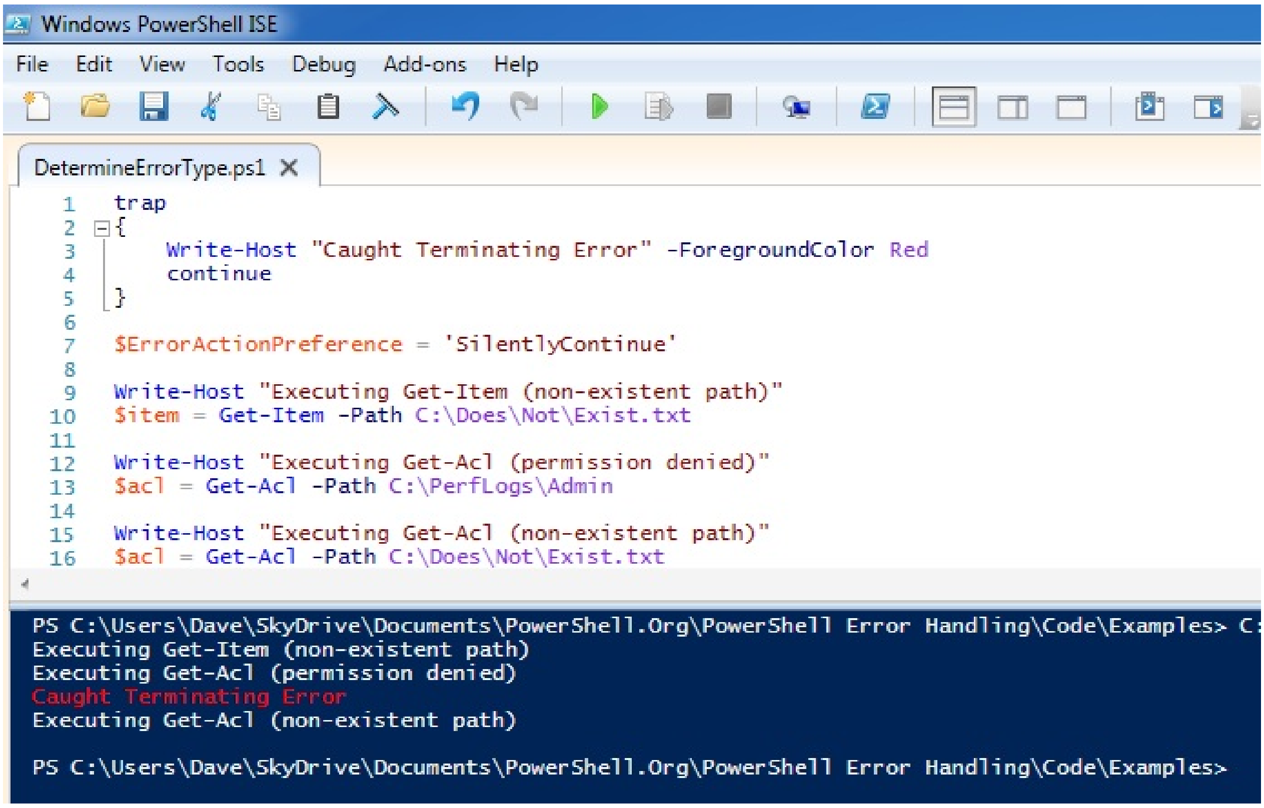


Figure 4.2: Identifying Terminating errors.

Ironically, this was a handy place both to use the Trap statement and to set $ErrorActionPreference to SilentlyContinue, both things that I would almost never do in an enterprise script. As you can see in figure 4.1, Get-Acl produces terminating exceptions when the file exists, but the cmdlet cannot read the ACL. Get-Item and Get-Acl both produce non-terminating errors if the file doesn't exist.

Going through this sort of trial and error can be a time-consuming process, though. You need to come up with the different ways a command might fail, and then reproduce those conditions to see if the resulting error was terminating or non-terminating. As a result of how annoying this can be, in addition to this ebook, the GitHub repository will contain a spreadsheet with a list of known Terminating errors from cmdlets. That will be a living document, possibly converted to a wiki at some point. While it will likely never be a complete reference, due to the massive number of PowerShell cmdlets out there, it's a lot better than nothing.

In addition to knowing whether errors are terminating or non-terminating, you may also want to know what types of Exceptions are being produced. Figure 4.3 demonstrates how you can list the exception types that are associated with different types of errors. Each Exception object may optionally contain an InnerException, and you can use any of them in a Catch or Trap block:

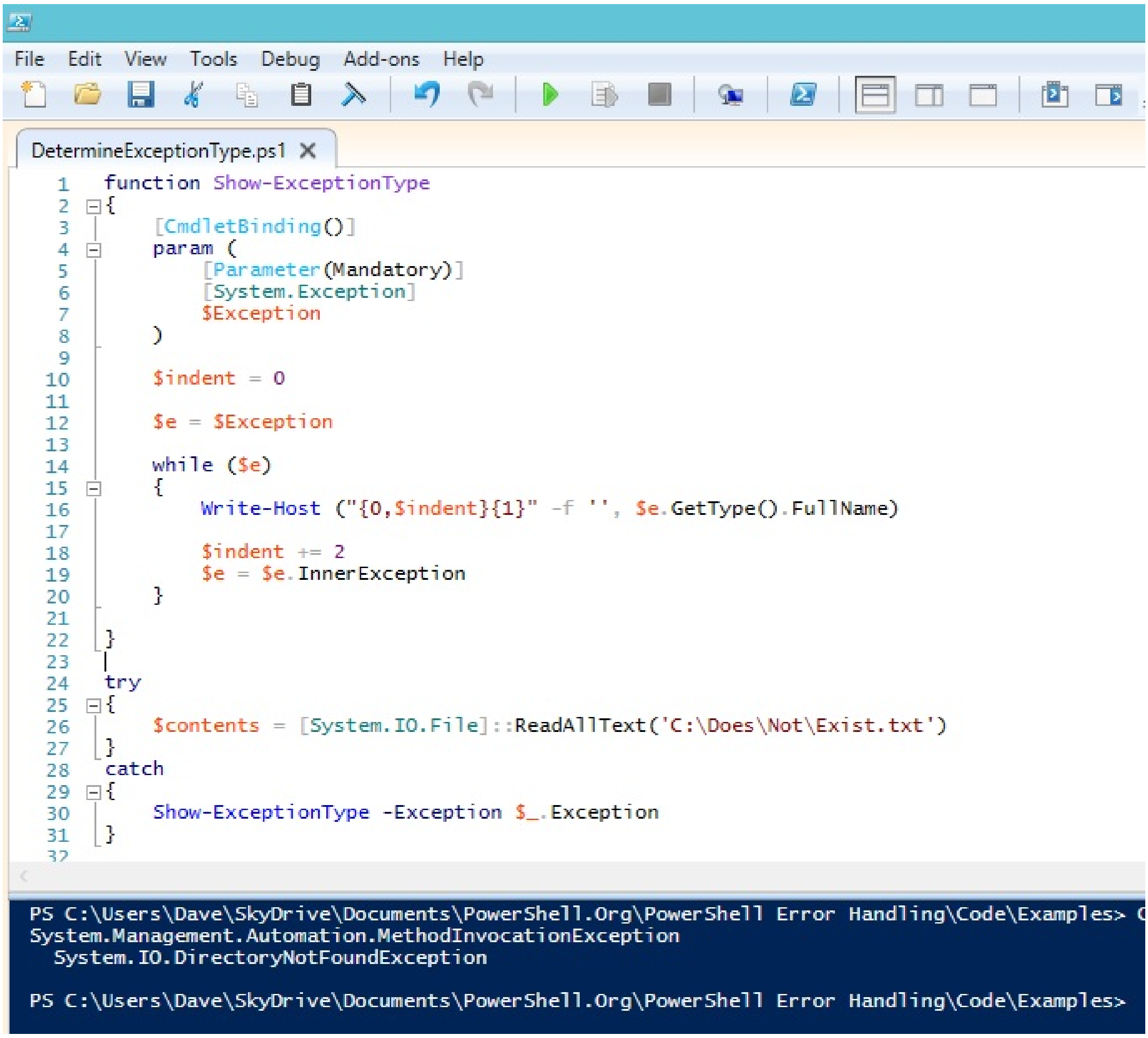


Figure 4.3: Displaying the types of Exceptions and any InnerExceptions.

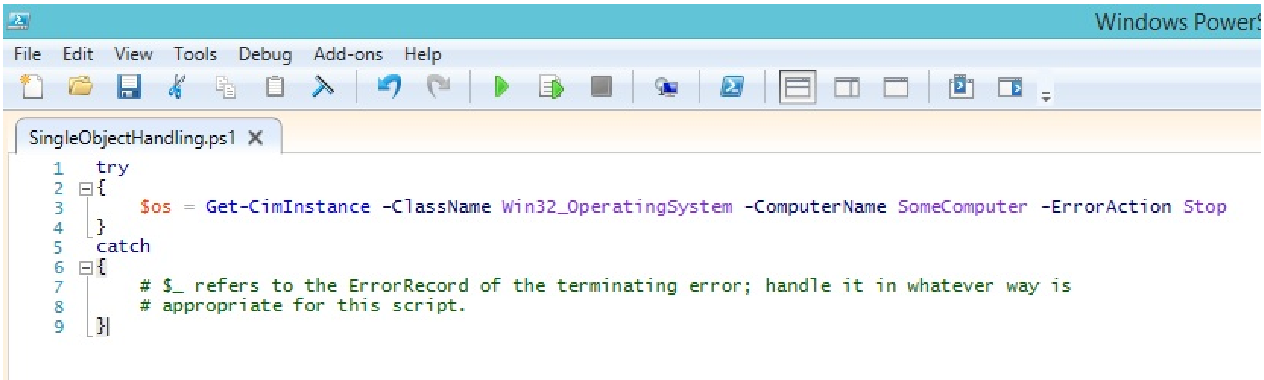
Dealing with Terminating Errors

This is the easy part. Just use Try/Catch, and refer to either $_ or $error[0] in your Catch blocks to get information about the terminating error.

Dealing with Non-Terminating Errors

I tend to categorize commands that can produce Non-Terminating errors (Cmdlets, functions and scripts) in one of three ways: Commands that only need to process a single input object, commands that can only produce Non-Terminating errors, and commands that could produce a Terminating or Non-Terminating error. I handle each of these categories in the following ways:

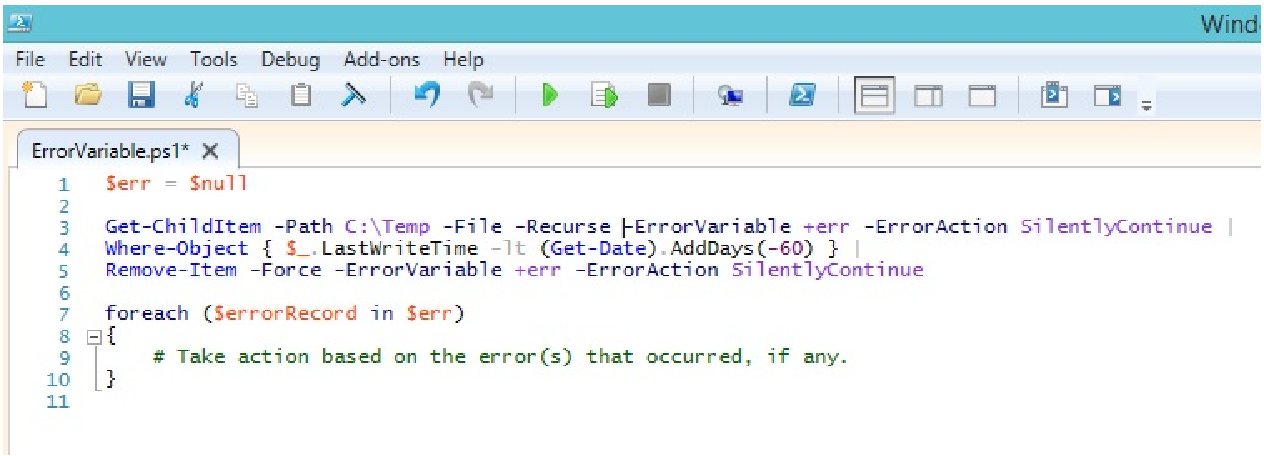
If the command only needs to process a single input object, as in figure 4.4, I use ErrorAction Stop and handle errors with Try/Catch. Because the cmdlet is only dealing with a single input object, the concept of a Non-Terminating error is not terribly useful anyway.




```
try
{
    $os = Get-CimInstance -ClassName Win32_OperatingSystem -ComputerName SomeComputer -ErrorAction Stop
}
catch
{
    # $_ refers to the ErrorRecord of the terminating error; handle it in whatever way is
    # appropriate for this script.
}
```

Figure 4.4: Using Try/Catch and ErrorAction Stop when dealing with a single object.

If the command should only ever produce Non-Terminating errors, I use ErrorVariable. This category is larger than you'd think; most PowerShell cmdlet errors are Non-Terminating:




```
$err = $null

Get-ChildItem -Path C:\Temp -File -Recurse -ErrorVariable +err -ErrorAction SilentlyContinue |
Where-Object { $_.LastWriteTime -lt (Get-Date).AddDays(-60) } |
Remove-Item -Force -ErrorVariable +err -ErrorAction SilentlyContinue

foreach ($errorRecord in $err)
{
    # Take action based on the error(s) that occurred, if any.
}
```

Figure 4.5: Using ErrorVariable when you won't be annoyed by its behavior arising from Terminating errors.

When you're examining the contents of your ErrorVariable, remember that you can usually get useful information about what failed by looking at an ErrorRecord's CategoryInfo.Activity property (which cmdlet produced the error) and TargetObject property (which object was it processing when the error occurred). However, not all cmdlets populate the ErrorRecord with a TargetObject, so you'll want to do some testing ahead of time to determine how useful this technique will be. If you find a situation where a cmdlet should be telling you about the TargetObject, but doesn't, consider changing your code structure to process one object at a time, as in figure 4.4. That way, you'll already know what object is being processed.

A trickier scenario arises if a particular command might produce either Terminating or Non-Terminating errors. In those situations, if it's practical, I try to change my code to call the command on one object at a time. If you find yourself in a situation where this is not

desirable (though I'm hard pressed to come up with an example), I recommend the following approach to avoid ErrorVariable's quirky behavior and also avoid calling $error.Clear():

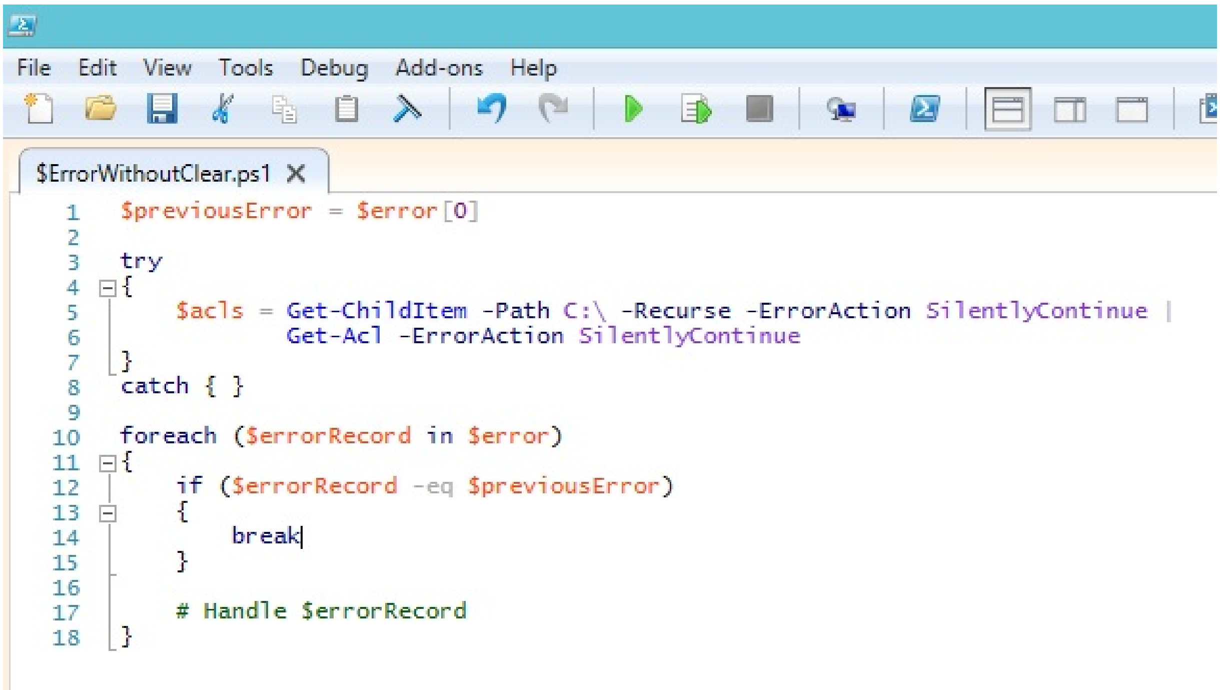


Figure 4.6: Using $error without calling Clear() and ignoring previously-existing error records.

As you can see, the structure of this code is almost the same as when using the ErrorVariable parameter, with the addition of a Try block around the offending code, and the use of the $previousError variable to make sure we're only reacting to new errors in the $error collection. In this case, I have an empty Catch block, because the terminating error (if one occurs) is going to be also added to $error and handled in the foreach loop anyway. You may prefer to handle the terminating error in the Catch block and non-terminating errors in the loop; either way works.

Calling external programs

When you need to call an external executable, most of the time you'll get the best results by checking $LASTEXITCODE for status information; however, you'll need do your homework on the program to make sure it returns useful information via its exit code. There are some odd executables out there that always return 0, regardless of whether they encountered errors.

If an external executable writes anything to the StdErr stream, PowerShell sometimes sees this and wraps the text in an ErrorRecord, but this behavior doesn't seem to be consistent. I'm not sure yet under what conditions these errors will be produced, so I tend to stick with $LASTEXITCODE when I need to tell whether an external command worked or not.

Afterword

We hope you've found this guide to be useful! This is always going to be a work in progress; as people bring material and suggestions, we'll incorporate that as best we can and publish a new edition.